1 Parser

Once the actual code has been separated from proof code, the Parser then parses into an abstract syntax tree.

```
module Parser where
  import Data.Map (Map)
  import qualified Data.Map.Strict as Map
```

The NativeAST is, for now, a placeholder for whatever type is produced by the lexer/parser of the language being proven.

```
  data NativeAST = NativeASTNode

  parseCode :: String → AST
  parseCode code = transformAST NativeASTNode

  transformAST :: NativeAST → AST
  transformAST native = ID "Hello␣world"
```

Once the code has been turned into a NativeAST, it is then transformed into the AST by the pluggable Transformer. Meanwhile, the proof code must also be converted into the definitions used to prove the program. These are represented by the same AST as the code, but this transformation is handled here.

```
  data AST = ID String -- name
           | Let AST AST AST -- ID Type Body
           | Type AST [AST] -- ID Parameters
           | Application AST [AST] -- Function Parameters
           | Function [AST] AST -- Parameters Body
           | Exists AST AST AST -- ID Type Body
           | TypeOf AST -- Type
           | Contradiction
           -- some value types, probably needed
           | VInteger Int -- Value
           | VDouble Double -- Value
           | VChar Char -- Value
           | VSymbol String -- For
           | VBoolean Bool -- True/False
           | VPair AST AST -- Head Tail

  parseProofs :: String → Map AST AST
  parseProofs proofs = Map.empty
```

Once parsing is complete the parsed definitions are placed into a map which, is then returned to the Compiler to be passed on to the Analyzer.